57

SOCKETS: UNIX DOMAIN

This chapter looks at the use of UNIX domain sockets, which allow communication between processes on the same host system. We discuss the use of both stream and datagram sockets in the UNIX domain. We also describe the use of file permissions to control access to UNIX domain sockets, the use of *socketpair()* to create a pair of connected UNIX domain sockets, and the Linux abstract socket namespace.

57.1 UNIX Domain Socket Addresses: *struct sockaddr_un*

In the UNIX domain, a socket address takes the form of a pathname, and the domain-specific socket address structure is defined as follows:

```
struct sockaddr_un {
    sa_family_t sun_family;         /* Always AF_UNIX */
    char sun_path[108];             /* Null-terminated socket pathname */
};
```

> The prefix *sun_* in the fields of the *sockaddr_un* structure has nothing to do with Sun Microsystems; rather, it derives from *socket unix*.

SUSv3 doesn't specify the size of the *sun_path* field. Early BSD implementations used 108 and 104 bytes, and one contemporary implementation (HP-UX 11) uses 92 bytes. Portable applications should code to this lower value, and use *snprintf()* or *strncpy()* to avoid buffer overruns when writing into this field.

In order to bind a UNIX domain socket to an address, we initialize a *sockaddr_un* structure, and then pass a (cast) pointer to this structure as the *addr* argument to *bind()*, and specify *addrlen* as the size of the structure, as shown in Listing 57-1.

Listing 57-1: Binding a UNIX domain socket

```
const char *SOCKNAME = "/tmp/mysock";
int sfd;
struct sockaddr_un addr;

sfd = socket(AF_UNIX, SOCK_STREAM, 0);          /* Create socket */
if (sfd == -1)
    errExit("socket");

memset(&addr, 0, sizeof(struct sockaddr_un));    /* Clear structure */
addr.sun_family = AF_UNIX;                       /* UNIX domain address */
strncpy(addr.sun_path, SOCKNAME, sizeof(addr.sun_path) - 1);

if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
    errExit("bind");
```

The use of the *memset()* call in Listing 57-1 ensures that all of the structure fields have the value 0. (The subsequent *strncpy()* call takes advantage of this by specifying its final argument as one less than the size of the *sun_path* field, to ensure that this field always has a terminating null byte.) Using *memset()* to zero out the entire structure, rather than initializing individual fields, ensures that any nonstandard fields that are provided by some implementations are also initialized to 0.

> The BSD-derived function *bzero()* is an alternative to *memset()* for zeroing the contents of a structure. SUSv3 specifies *bzero()* and the related *bcopy()* (which is similar to *memmove()*), but marks both functions LEGACY, noting that *memset()* and *memmove()* are preferred. SUSv4 removes the specifications of *bzero()* and *bcopy()*.

When used to bind a UNIX domain socket, *bind()* creates an entry in the file system. (Thus, a directory specified as part of the socket pathname needs to be accessible and writable.) The ownership of the file is determined according to the usual rules for file creation (Section 15.3.1). The file is marked as a socket. When *stat()* is applied to this pathname, it returns the value `S_IFSOCK` in the file-type component of the *st_mode* field of the *stat* structure (Section 15.1). When listed with *ls –l*, a UNIX domain socket is shown with the type *s* in the first column, and *ls –F* appends an equal sign (=) to the socket pathname.

> Although UNIX domain sockets are identified by pathnames, I/O on these sockets doesn't involve operations on the underlying device.

The following points are worth noting about binding a UNIX domain socket:

- We can't bind a socket to an existing pathname (*bind()* fails with the error `EADDRINUSE`).

- It is usual to bind a socket to an absolute pathname, so that the socket resides at a fixed address in the file system. Using a relative pathname is possible, but unusual, because it requires an application that wants to *connect()* to this socket to know the current working directory of the application that performs the *bind()*.
- A socket may be bound to only one pathname; conversely, a pathname can be bound to only one socket.
- We can't use *open()* to open a socket.
- When the socket is no longer required, its pathname entry can (and generally should) be removed using *unlink()* (or *remove()*).

In most of our example programs, we bind UNIX domain sockets to pathnames in the `/tmp` directory, because this directory is normally present and writable on every system. This makes it easy for the reader to run these programs without needing to first edit the socket pathnames. Be aware, however, that this is generally not a good design technique. As pointed out in Section 38.7, creating files in publicly writable directories such as `/tmp` can lead to various security vulnerabilities. For example, by creating a pathname in `/tmp` with the same name as that used by the application socket, we can create a simple denial-of-service attack. Real-world applications should *bind()* UNIX domain sockets to absolute pathnames in suitably secured directories.

57.2 Stream Sockets in the UNIX Domain

We now present a simple client-server application that uses stream sockets in the UNIX domain. The client program (Listing 57-4) connects to the server, and uses the connection to transfer data from its standard input to the server. The server program (Listing 57-3) accepts client connections, and transfers all data sent on the connection by the client to standard output. The server is a simple example of an *iterative* server—a server that handles one client at a time before proceeding to the next client. (We consider server design in more detail in Chapter 60.)

Listing 57-2 is the header file used by both of these programs.

Listing 57-2: Header file for `us_xfr_sv.c` and `us_xfr_cl.c`

sockets/us_xfr.h

```
#include <sys/un.h>
#include <sys/socket.h>
#include "tlpi_hdr.h"

#define SV_SOCK_PATH "/tmp/us_xfr"

#define BUF_SIZE 100
```

sockets/us_xfr.h

In the following pages, we first present the source code of the server and client, and then discuss the details of these programs and show an example of their use.

Listing 57-3: A simple UNIX domain stream socket server

sockets/us_xfr_sv.c

```
#include "us_xfr.h"

#define BACKLOG 5

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        errExit("socket");

    /* Construct server socket address, bind socket to it,
       and make this a listening socket */

    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);

    if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");

    if (listen(sfd, BACKLOG) == -1)
        errExit("listen");

    for (;;) {          /* Handle client connections iteratively */

        /* Accept a connection. The connection is returned on a new
           socket, 'cfd'; the listening socket ('sfd') remains open
           and can be used to accept further connections. */

        cfd = accept(sfd, NULL, NULL);
        if (cfd == -1)
            errExit("accept");

        /* Transfer data from connected socket to stdout until EOF */

        while ((numRead = read(cfd, buf, BUF_SIZE)) > 0)
            if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("partial/failed write");

        if (numRead == -1)
            errExit("read");
        if (close(cfd) == -1)
            errMsg("close");
    }
}
```
sockets/us_xfr_sv.c

Listing 57-4: A simple UNIX domain stream socket client

sockets/us_xfr_cl.c
```
#include "us_xfr.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);       /* Create client socket */
    if (sfd == -1)
        errExit("socket");

    /* Construct server address, and make the connection */

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);

    if (connect(sfd, (struct sockaddr *) &addr,
                sizeof(struct sockaddr_un)) == -1)
        errExit("connect");

    /* Copy stdin to socket */

    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
        if (write(sfd, buf, numRead) != numRead)
            fatal("partial/failed write");

    if (numRead == -1)
        errExit("read");

    exit(EXIT_SUCCESS);         /* Closes our socket; server sees EOF */
}
```
sockets/us_xfr_cl.c

The server program is shown in Listing 57-3. The server performs the following steps:

- Create a socket.
- Remove any existing file with the same pathname as that to which we want to bind the socket.

- Construct an address structure for the server's socket, bind the socket to that address, and mark the socket as a listening socket.
- Execute an infinite loop to handle incoming client requests. Each loop iteration performs the following steps:
  - Accept a connection, obtaining a new socket, *cfd*, for the connection.
  - Read all of the data from the connected socket and write it to standard output.
  - Close the connected socket *cfd*.

The server must be terminated manually (e.g., by sending it a signal).

The client program (Listing 57-4) performs the following steps:

- Create a socket.
- Construct the address structure for the server's socket and connect to the socket at that address.
- Execute a loop that copies its standard input to the socket connection. Upon encountering end-of-file in its standard input, the client terminates, with the result that its socket is closed and the server sees end-of-file when reading from the socket on the other end of the connection.

The following shell session log demonstrates the use of these programs. We begin by running the server in the background:

```
$ ./us_xfr_sv > b &
[1] 9866
$ ls -lF /tmp/us_xfr                          Examine socket file with ls
srwxr-xr-x    1 mtk      users         0 Jul 18 10:48 /tmp/us_xfr=
```

We then create a test file to be used as input for the client, and run the client:

```
$ cat *.c > a
$ ./us_xfr_cl < a                             Client takes input from test file
```

At this point, the child has completed. Now we terminate the server as well, and check that the server's output matches the client's input:

```
$ kill %1                                     Terminate server
 [1]+  Terminated   ./us_xfr_sv >b            Shell sees server's termination
$ diff a b
$
```

The *diff* command produces no output, indicating that the input and output files are identical.

Note that after the server terminates, the socket pathname continues to exist. This is why the server uses *remove()* to remove any existing instance of the socket pathname before calling *bind()*. (Assuming we have appropriate permissions, this *remove()* call would remove any type of file with this pathname, even if it wasn't a socket.) If we did not do this, then the *bind()* call would fail if a previous invocation of the server had already created this socket pathname.

57.3 Datagram Sockets in the UNIX Domain

In the generic description of datagram sockets that we provided in Section 56.6, we stated that communication using datagram sockets is unreliable. This is the case for datagrams transferred over a network. However, for UNIX domain sockets, datagram transmission is carried out within the kernel, and is reliable. All messages are delivered in order and unduplicated.

Maximum datagram size for UNIX domain datagram sockets

SUSv3 doesn't specify a maximum size for datagrams sent via a UNIX domain socket. On Linux, we can send quite large datagrams. The limits are controlled via the `SO_SNDBUF` socket option and various `/proc` files, as described in the *socket(7)* manual page. However, some other UNIX implementations impose lower limits, such as 2048 bytes. Portable applications employing UNIX domain datagram sockets should consider imposing a low upper limit on the size of datagrams used.

Example program

Listing 57-6 and Listing 57-7 show a simple client-server application using UNIX domain datagram sockets. Both of these programs make use of the header file shown in Listing 57-5.

Listing 57-5: Header file used by `ud_ucase_sv.c` and `ud_ucase_cl.c`

```
–––––––––––––––––––––––––––––––––––––––––––––––––––––– sockets/ud_ucase.h
#include <sys/un.h>
#include <sys/socket.h>
#include <ctype.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 10             /* Maximum size of messages exchanged
                                   between client to server */

#define SV_SOCK_PATH "/tmp/ud_ucase"
–––––––––––––––––––––––––––––––––––––––––––––––––––––– sockets/ud_ucase.h
```

The server program (Listing 57-6) first creates a socket and binds it to a well-known address. (Beforehand, the server unlinks the pathname matching that address, in case the pathname already exists.) The server then enters an infinite loop, using *recvfrom()* to receive datagrams from clients, converting the received text to uppercase, and returning the converted text to the client using the address obtained via *recvfrom()*.

The client program (Listing 57-7) creates a socket and binds the socket to an address, so that the server can send its reply. The client address is made unique by including the client's process ID in the pathname. The client then loops, sending each of its command-line arguments as a separate message to the server. After sending each message, the client reads the server response and displays it on standard output.

Listing 57-6: A simple UNIX domain datagram server

sockets/ud_ucase_sv.c

```
#include "ud_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un svaddr, claddr;
    int sfd, j;
    ssize_t numBytes;
    socklen_t len;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_DGRAM, 0);       /* Create server socket */
    if (sfd == -1)
        errExit("socket");

    /* Construct well-known address and bind server socket to it */

    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);

    memset(&svaddr, 0, sizeof(struct sockaddr_un));
    svaddr.sun_family = AF_UNIX;
    strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(svaddr.sun_path) - 1);

    if (bind(sfd, (struct sockaddr *) &svaddr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");

    /* Receive messages, convert to uppercase, and return to client */

    for (;;) {
        len = sizeof(struct sockaddr_un);
        numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                            (struct sockaddr *) &claddr, &len);
        if (numBytes == -1)
            errExit("recvfrom");

        printf("Server received %ld bytes from %s\n", (long) numBytes,
                claddr.sun_path);

        for (j = 0; j < numBytes; j++)
            buf[j] = toupper((unsigned char) buf[j]);

        if (sendto(sfd, buf, numBytes, 0, (struct sockaddr *) &claddr, len) !=
                numBytes)
            fatal("sendto");
    }
}
```

sockets/ud_ucase_sv.c

Listing 57-7: A simple UNIX domain datagram client

sockets/ud_ucase_cl.c

```
#include "ud_ucase.h"

int
main(int argc, char *argv[])
{
    struct sockaddr_un svaddr, claddr;
    int sfd, j;
    size_t msgLen;
    ssize_t numBytes;
    char resp[BUF_SIZE];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s msg...\n", argv[0]);

    /* Create client socket; bind to unique pathname (based on PID) */

    sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sfd == -1)
        errExit("socket");

    memset(&claddr, 0, sizeof(struct sockaddr_un));
    claddr.sun_family = AF_UNIX;
    snprintf(claddr.sun_path, sizeof(claddr.sun_path),
            "/tmp/ud_ucase_cl.%ld", (long) getpid());

    if (bind(sfd, (struct sockaddr *) &claddr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");

    /* Construct address of server */

    memset(&svaddr, 0, sizeof(struct sockaddr_un));
    svaddr.sun_family = AF_UNIX;
    strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(svaddr.sun_path) - 1);

    /* Send messages to server; echo responses on stdout */

    for (j = 1; j < argc; j++) {
        msgLen = strlen(argv[j]);       /* May be longer than BUF_SIZE */
        if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
                sizeof(struct sockaddr_un)) != msgLen)
            fatal("sendto");

        numBytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
        if (numBytes == -1)
            errExit("recvfrom");
        printf("Response %d: %.*s\n", j, (int) numBytes, resp);
    }

    remove(claddr.sun_path);            /* Remove client socket pathname */
    exit(EXIT_SUCCESS);
}
```

sockets/ud_ucase_cl.c

The following shell session log demonstrates the use of the server and client programs:

```
$ ./ud_ucase_sv &
[1] 20113
$ ./ud_ucase_cl hello world                Send 2 messages to server
Server received 5 bytes from /tmp/ud_ucase_cl.20150
Response 1: HELLO
Server received 5 bytes from /tmp/ud_ucase_cl.20150
Response 2: WORLD
$ ./ud_ucase_cl 'long message'             Send 1 longer message to server
Server received 10 bytes from /tmp/ud_ucase_cl.20151
Response 1: LONG MESSA
$ kill %1                                  Terminate server
```

The second invocation of the client program was designed to show that when a *recvfrom()* call specifies a *length* (`BUF_SIZE`, defined in Listing 57-5 with the value 10) that is shorter than the message size, the message is silently truncated. We can see that this truncation occurred, because the server prints a message saying it received just 10 bytes, while the message sent by the client consisted of 12 bytes.

57.4 UNIX Domain Socket Permissions

The ownership and permissions of the socket file determine which processes are able to communicate with that socket:

- To connect to a UNIX domain stream socket, write permission is required on the socket file.
- To send a datagram to a UNIX domain datagram socket, write permission is required on the socket file.

In addition, execute (search) permission is required on each of the directories in the socket pathname.

By default, a socket is created (by *bind()*) with all permissions granted to owner (user), group, and other. To change this, we can precede the call to *bind()* with a call to *umask()* to disable the permissions that we do not wish to grant.

Some systems ignore the permissions on the socket file (SUSv3 allows this). Thus, we can't portably use socket file permissions to control access to the socket, although we can portably use permissions on the hosting directory for this purpose.

57.5 Creating a Connected Socket Pair: *socketpair()*

Sometimes, it is useful for a single process to create a pair of sockets and connect them together. This could be done using two calls to *socket()*, a call to *bind()*, and then either calls to *listen()*, *connect()*, and *accept()* (for stream sockets), or a call to *connect()* (for datagram sockets). The *socketpair()* system call provides a shorthand for this operation.

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sockfd[2]);
                                    Returns 0 on success, or –1 on error
```

This *socketpair()* system call can be used only in the UNIX domain; that is, *domain* must be specified as AF_UNIX. (This restriction applies on most implementations, and is logical, since the socket pair is created on a single host system.) The socket *type* may be specified as either SOCK_DGRAM or SOCK_STREAM. The *protocol* argument must be specified as 0. The *sockfd* array returns the file descriptors referring to the two connected sockets.

Specifying *type* as SOCK_STREAM creates the equivalent of a bidirectional pipe (also known as a *stream pipe*). Each socket can be used for both reading and writing, and separate data channels flow in each direction between the two sockets. (On BSD-derived implementations, *pipe()* is implemented as a call to *socketpair()*.)

Typically, a socket pair is used in a similar fashion to a pipe. After the *socketpair()* call, the process then creates a child via *fork()*. The child inherits copies of the parent's file descriptors, including the descriptors referring to the socket pair. Thus, the parent and child can use the socket pair for IPC.

One way in which the use of *socketpair()* differs from creating a pair of connected sockets manually is that the sockets are not bound to any address. This can help us avoid a whole class of security vulnerabilities, since the sockets are not visible to any other process.

> Starting with kernel 2.6.27, Linux provides a second use for the *type* argument, by allowing two nonstandard flags to be ORed with the socket type. The SOCK_CLOEXEC flag causes the kernel to enable the close-on-exec flag (FD_CLOEXEC) for the two new file descriptors. This flag is useful for the same reasons as the *open()* O_CLOEXEC flag described in Section 4.3.1. The SOCK_NONBLOCK flag causes the kernel to set the O_NONBLOCK flag on both underlying open file descriptions, so that future I/O operations on the socket will be nonblocking. This saves additional calls to *fcntl()* to achieve the same result.

57.6 The Linux Abstract Socket Namespace

The so-called *abstract namespace* is a Linux-specific feature that allows us to bind a UNIX domain socket to a name without that name being created in the file system. This provides a few potential advantages:

- We don't need to worry about possible collisions with existing names in the file system.
- It is not necessary to unlink the socket pathname when we have finished using the socket. The abstract name is automatically removed when the socket is closed.
- We don't need to create a file-system pathname for the socket. This may be useful in a *chroot* environment, or if we don't have write access to a file system.

To create an abstract binding, we specify the first byte of the *sun_path* field as a null byte (`\0`). This distinguishes abstract socket names from conventional UNIX domain socket pathnames, which consist of a string of one or more nonnull bytes terminated by a null byte. The remaining bytes of the *sun_path* field then define the abstract name for the socket. These bytes are interpreted in their entirety, rather than as a null-terminated string.

Listing 57-8 demonstrates the creation of an abstract socket binding.

Listing 57-8: Creating an abstract socket binding

from **sockets/us_abstract_bind.c**

```
    struct sockaddr_un addr;

    memset(&addr, 0, sizeof(struct sockaddr_un));  /* Clear address structure */
    addr.sun_family = AF_UNIX;                     /* UNIX domain address */

    /* addr.sun_path[0] has already been set to 0 by memset() */

    strncpy(&addr.sun_path[1], "xyz", sizeof(addr.sun_path) - 2);
                /* Abstract name is "xyz" followed by null bytes */

    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sockfd == -1)
        errExit("socket");

    if (bind(sockfd, (struct sockaddr *) &addr,
            sizeof(struct sockaddr_un)) == -1)
        errExit("bind");
```

from **sockets/us_abstract_bind.c**

The fact that an initial null byte is used to distinguish an abstract socket name from a conventional socket name can have an unusual consequence. Suppose that the variable *name* happens to point to a zero-length string and that we attempt to bind a UNIX domain socket to a *sun_path* initialized as follows:

```
    strncpy(addr.sun_path, name, sizeof(addr.sun_path) - 1);
```

On Linux, we'll inadvertently create an abstract socket binding. However, such a code sequence is probably unintentional (i.e., a bug). On other UNIX implementations, the subsequent *bind()* would fail.

57.7 Summary

UNIX domain sockets allow communication between applications on the same host. The UNIX domain supports both stream and datagram sockets.

A UNIX domain socket is identified by a pathname in the file system. File permissions can be used to control access to a UNIX domain socket.

The *socketpair()* system call creates a pair of connected UNIX domain sockets. This avoids the need for multiple system calls to create, bind, and connect the sockets. A socket pair is normally used in a similar fashion to a pipe: one process creates the socket pair and then forks to create a child that inherits descriptors referring to the sockets. The two processes can then communicate via the socket pair.

The Linux-specific abstract socket namespace allows us to bind a UNIX domain socket to a name that doesn't appear in the file system.

Further information

Refer to the sources of further information listed in Section 59.15.

57.8 Exercises

57-1. In Section 57.3, we noted that UNIX domain datagram sockets are reliable. Write programs to show that if a sender transmits datagrams to a UNIX domain datagram socket faster than the receiver reads them, then the sender is eventually blocked, and remains blocked until the receiver reads some of the pending datagrams.

57-2. Rewrite the programs in Listing 57-3 (`us_xfr_sv.c`) and Listing 57-4 (`us_xfr_cl.c`) to use the Linux-specific abstract socket namespace (Section 57.6).

57-3. Reimplement the sequence-number server and client of Section 44.8 using UNIX domain stream sockets.

57-4. Suppose that we create two UNIX domain datagram sockets bound to the paths `/somepath/a` and `/somepath/b`, and that we connect the socket `/somepath/a` to `/somepath/b`. What happens if we create a third datagram socket and try to send (*sendto()*) a datagram via that socket to `/somepath/a`? Write a program to determine the answer. If you have access to other UNIX systems, test the program on those systems to see if the answer differs.